Linux Kernel Module Programming

I

...

Sistemas de Computación (IComp)

¿ Cómo empezamos con esto ?

Arquitectura interna de Linux (2016)
Dr. Juan Carlos Sáez Alcaide
(U Complutense de Madrid - link)

Facultad de Informática de la UN La Plata

¿ Módulos de Kernel ?

El kernel de Linux es modular : permite insertar y eliminar código bajo demanda con el fin de añadir o quitar una funcionalidad. Funcionalidades que podemos añadir con módulos:

- Drivers privativos de hardware para gráficos.
- Registrar temperaturas de componentes del ordenador.
- Hacer funcionar la tarjeta de red wifi.
- Etc.

Básicamente los módulos del kernel nos permiten hacer funcionar el hardware, y si funciona, podemos hacer que lo haga de forma más eficiente aún.-

Manos a la obra.!!

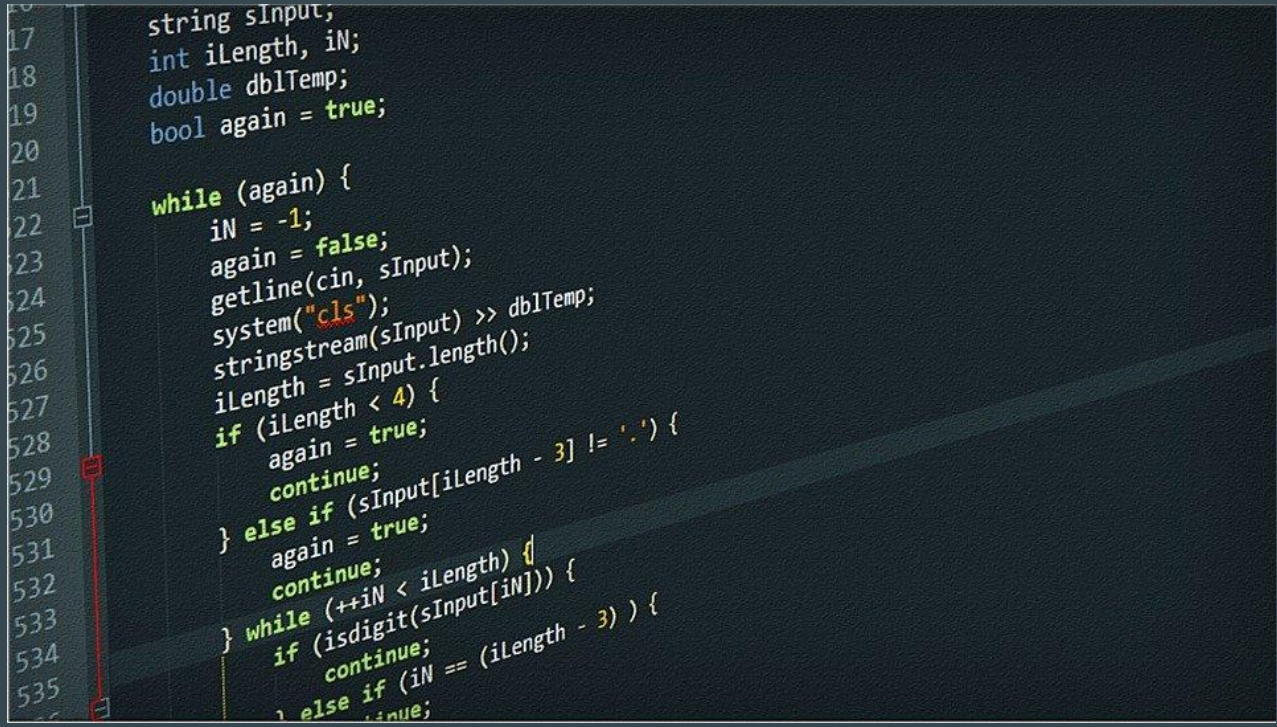
1) Instalar package como para llegar a compilar el kernel completo.
2) Descargar los primeros ejemplos del curso de Juan Carlos.
3) Compilar los ejemplos.
4) Insertar / Remover los módulos.

Sacar algunas conclusiones..!! documentar todo en el informe...

Primeras tareas

```
$ sudo apt-get install build-essentials kernel-package
$ sudo apt-get install linux-source
$ echo Estamos listos  para compilar nuestro primer módulo..!!
```

Mi primer módulo



Explicar fuente desde fuente..!!

Mi primer módulo

```
$ make
$ make clean
$ sudo insmod mimodulo.ko
$ sudo lsmod | head
$ sudo lsmod | grep mimodulo
$ dmesg | tail
$ sudo rmmod mimodulo
$ dmesg | tail

$ modinfo mimodulo.ko
```

Mi primer módulo

¿ Profe... y qué hace este módulo ?

El sistema de ficheros /proc

Contiene un sistema de ficheros virtual. No existe físicamente en disco, sino que el kernel lo crea en memoria. Se utiliza para ofrecer información relacionada con el sistema (originalmente acerca de procesos, de aquí su nombre). Algunos de esos ficheros :

/proc/cpuinfo; Información acerca del procesador: tipo, marca, modelo, rendimiento, etc. (lscpu)

/proc/modules; Indica los módulos del núcleo que han sido cargados hasta el momento. (lsmod)

El sistema de ficheros /proc

Podemos consultar mayor información en este link:

5.2. TOP-LEVEL FILES WITHIN THE PROC FILE SYSTEM

¿ De qué módulos dispone mi kernel ?

```
$ ls -l /lib/modules/$(uname -r)/kernel

$ lspci -v
$ lsmod | grep pci-module-info

$ lsusb -v
$ lsmod | grep usb-module-info
```

https://linux-hardware.org/

```
$ sudo -E hw-probe -all -upload
```

Firma de módulos

Podemos consultar mayor información sobre cómo firmar un módulo en este link:

3.12. FIRMA DE MÓDULOS DEL KERNEL PARA EL ARRANQUE SEGURO

https://ubuntu.com/blog/how-to-sign-things-for-secure-boot

Desafío #1: Mejorar la seguridad del kernel

Revisar el link anterior para impulsar acciones que permitan mejorar la seguridad del kernel, concretamente: evitando cargar módulos que no estén firmados.
¿Qué otras medidas de seguridad podemos implementar para asegurar nuestro ordenador desde el kernel ?

Desafío #2: Módulos vs Programas

1. ¿ Cómo empiezan y terminan unos y otros ?
2. ¿ Qué funciones tiene disponible un programa y un módulo ?
3. Espacio de usuario vs espacio del kernel.
4. Espacio de datos y espacio de código para usuario y kernel. Riesgos.!!
5. Drivers. Investigar contenido de /dev.

Puede consultar esta referencia:

The Linux Kernel Module Programming Guide. Peter Jay Salzman, Michael Burian, Ori Pomerantz, Bob Mottram, Jim Huang. April 28, 2022.-

Desafío #2: Módulos vs programas

¿ Cómo empiezan y terminan unos y otros ?

Un programa comienza con una función main(), se ejecuta hasta la última línea de código y retorna el control al sistema operativo o a quien los convocó.

Un módulo no.!!

Un módulo comienza con la ejecución de una función module_init(), le dice al kernel qué funcionalidad ofrece y setea el kernel a la espera de que lo utilice cuando el kernel lo necesite. Los módulos deben tener una función de entrada y otra de salida (module_exit())

Desafío #2: Módulos vs programas

¿ Qué funciones tiene disponible un programa ?

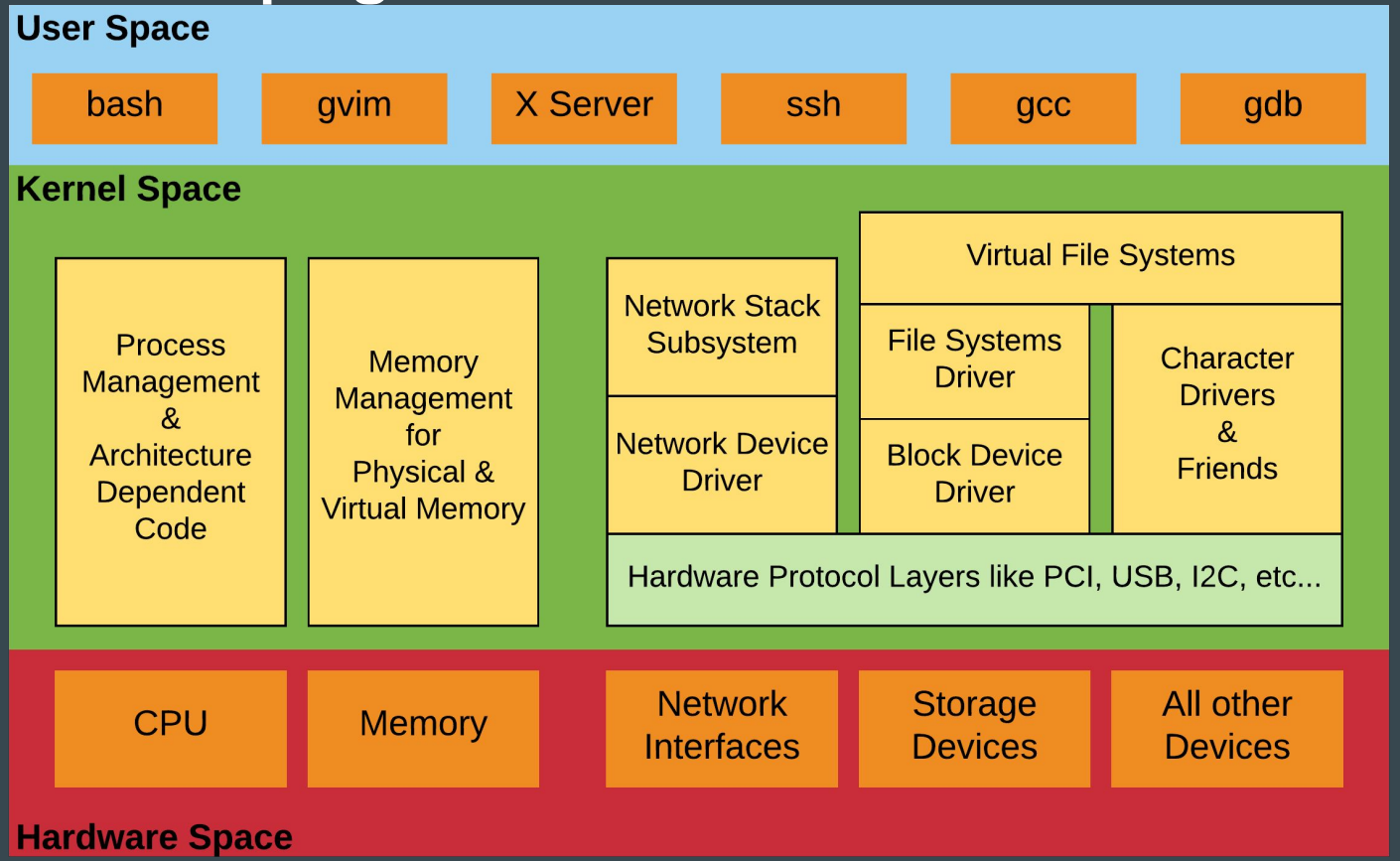
Primero construir un programa que llame a printf(), compilarlo (con opción -Wall y ejecutar strace (strace -tt y strace -c). Explicar lo que se observa.!!

¿ Qué funciones tiene disponible un módulo ?

La definición de los símbolos proviene del propio kernel; las únicas funciones externas que puede utilizar un módulo, son las proporcionadas por el kernel. Investigar el contenido del archivo /proc/kallsyms .!!

Desafío #2: Módulos vs programas

Espacio de usuario vs espacio del kernel



Desafío #2: Módulos vs programas

User Space vs Kernel Space

Así como cada proceso tiene su espacio de memoria asignado, el kernel tiene su propio espacio de memoria. Y dado que los módulos pasan a formar parte del espacio de kernel, cuando fallan, falla el kernel.!!

Desafío #2: Módulos vs programas

Name Space

Un programita en C, utiliza nombres de variables convenientes y con sentido para el lector. Ahora, si escribe rutinas que serán parte de un gran programa, cualquier variable global que tenga es parte de una comunidad de variables globales de otras personas y algunos de los nombres pueden entrar en conflicto. Cuando existen muchas variables globales que no son lo suficientemente significativas para ser distinguidas, se contamina el espacio de nombres. En grandes proyectos, se debe hacer un esfuerzo para recordar nombres reservados y encontrar formas de desarrollar un esquema para nombres y símbolos únicos de variables.

El archivo /proc/kallsyms contiene todos los símbolos que el kernel conoce y que, por lo tanto, son accesibles para sus módulos, ya que comparten el espacio de código del kernel.

Desafío #2: Módulos vs programas

Drivers. Investigar contenido de /dev.

Una clase particular de módulo es el “driver” o controlador de dispositivo, que proporciona funcionalidad para hardware específico.

Como en los sistemas “unix” los dispositivos se mapean como archivos, los vamos a encontrar asociados con alguna entrada en /dev.

Observaremos de forma general, qué tipos existen y prestaremos atención a los números mayor y menor.

¿ Quién carga los módulos del kernel ?

El kernel de Linux tiene un diseño modular. En el momento de arranque, sólo se carga un kernel residente mínimo en memoria. Cuando un usuario solicita alguna característica que no está presente en el kernel residente, se carga dinámicamente en memoria un módulo kernel, también conocido algunas veces como un driver.

Durante la instalación, se prueba el hardware en el sistema. Basado en esta prueba y en la información proporcionada por el usuario, el programa de instalación decide qué módulos necesita cargar en el momento de arranque. El programa de instalación configura el mecanismo de carga dinámica para que funcione de forma transparente.

¿ Quién carga los módulos del kernel ?

Si se añade un nuevo hardware después de la instalación y este requiere un módulo kernel, el sistema debe ser configurado para cargar el módulo adecuado para el nuevo hardware.

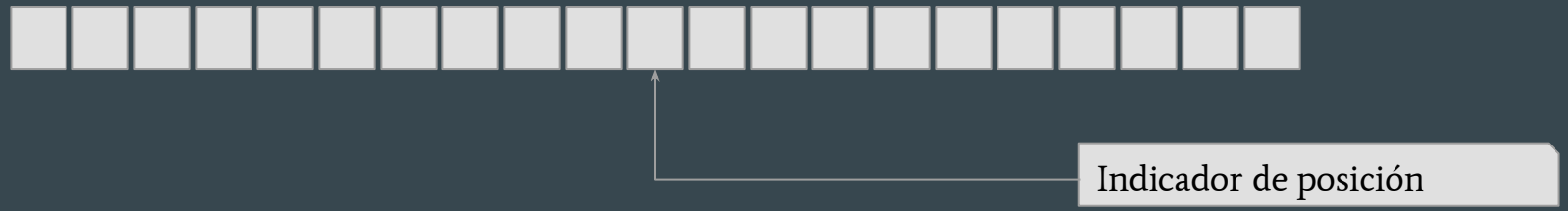
Cuando el sistema es arrancado con el nuevo hardware, se ejecuta el programa Kudzu (daemon hardware autodetection) que detecta el nuevo hardware si es soportado y configura el módulo necesario para él.

El módulo también puede ser especificado manualmente modificando el archivo de configuración del módulo, /etc/modules.conf.

API de Linux para acceso a ficheros

API de Linux para acceso a ficheros

¿ Qué nos ofrece la API de Linux para gestionar ficheros ?



- open(): Abre el fichero a partir de su ruta y devuelve un descriptor de fichero (int)
- read(), write(): Para leer/escribir en el fichero avanzando el puntero de posición de forma implícita.
- lseek(): Mover el puntero de posición, explícitamente a un lugar.
- close(): Cerrar el fichero a partir de su descriptor.

open() (link)

```
int open(const char *pathname, int flags, mode_t mode);
```

Abre el archivo al que apunta pathname y asocia un streaming con él. Los flags deben incluir uno de los siguientes modos de acceso: O_RDONLY, O_WRONLY o O_RDWR. Estos solicitan abrir el archivo de solo lectura, solo escritura o lectura/escritura, respectivamente. El mode se refiere a los privilegios con los que se abrirá el archivo para el usuario, para el grupo y para el resto del mundo.

Devuelve un descriptor de archivo, numérico o (-1) en caso de error.

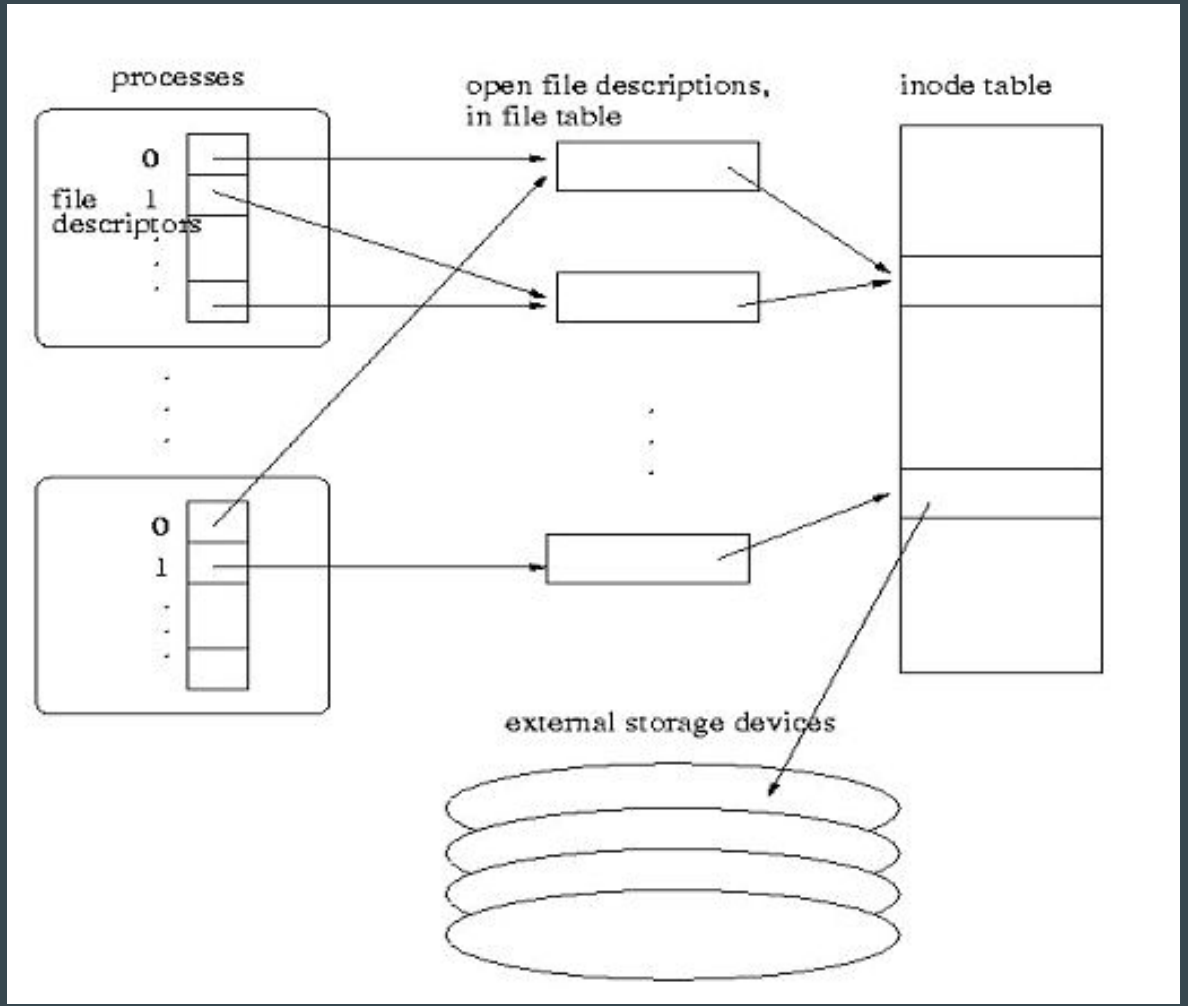
close()

(link)

```
int close(int fd);
```

Cierra un archivo a partir de su descriptor fd. Retorna cero en caso de éxito y -1 en caso de algún error.

Estructura de los datos en un sistema de archivos



read() (link)

```
ssize_t read(int fd, void *buf, size_t n_bytes);
```

Transfiere hasta n_bytes del fichero a la región de memoria del proceso (buf) y avanza el puntero de posición del fichero (tantos bytes como lee).

Argumentos:

- fd: Descriptor del fichero (int)
- buf: Puntero a la región de memoria de donde el SO leerá los datos
- n_bytes: Número de bytes a leer

Retorno:

- En caso de éxito, retorna el número de bytes escritos ó 0 (EOF).
- En caso de fallo, la función retorna -1

write()

```
ssize_t write(int fd, void *buf, size_t n_bytes);
```

Transfiere hasta n_bytes de la región de memoria del proceso (buf) al fichero y avanza implícitamente su puntero de posición tantos bytes como se escriban.

Argumentos:

- fd: Descriptor del fichero (int)
- buf: Puntero a la región de memoria donde se copiaran los datos
- n_bytes: Número de bytes a escribir

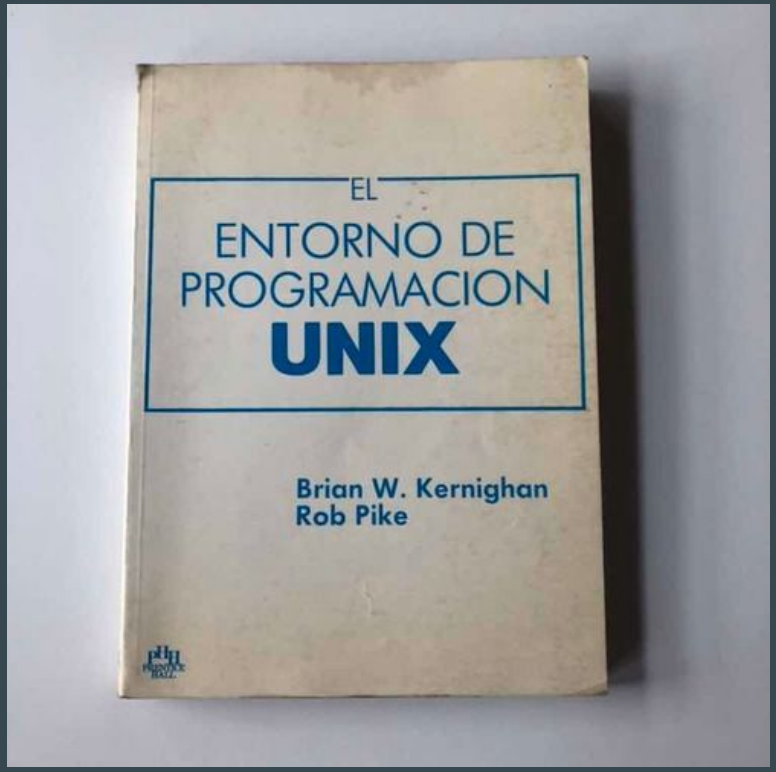
Retorno:

- En caso de éxito, retorna el número de bytes leídos.
- En caso de fallo, la función retorna -1

Esto tiene sus años..!!

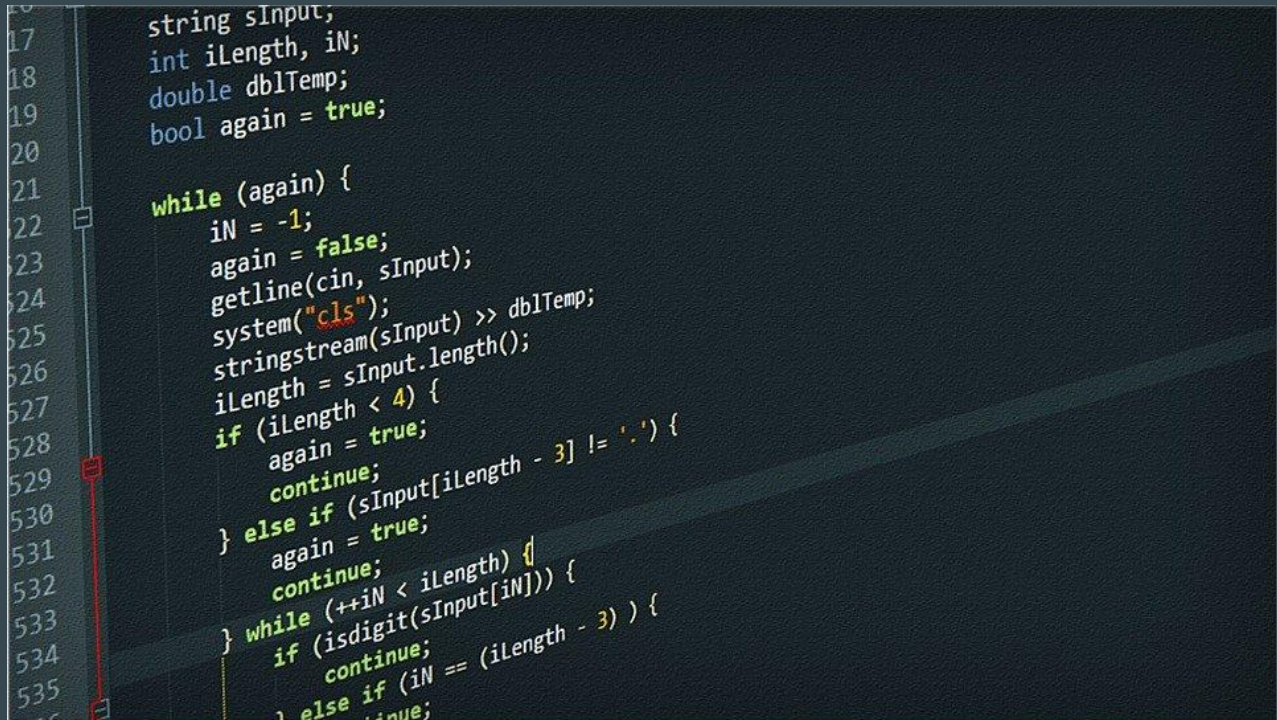
1987, Primer edición en español..!!

File System (link)



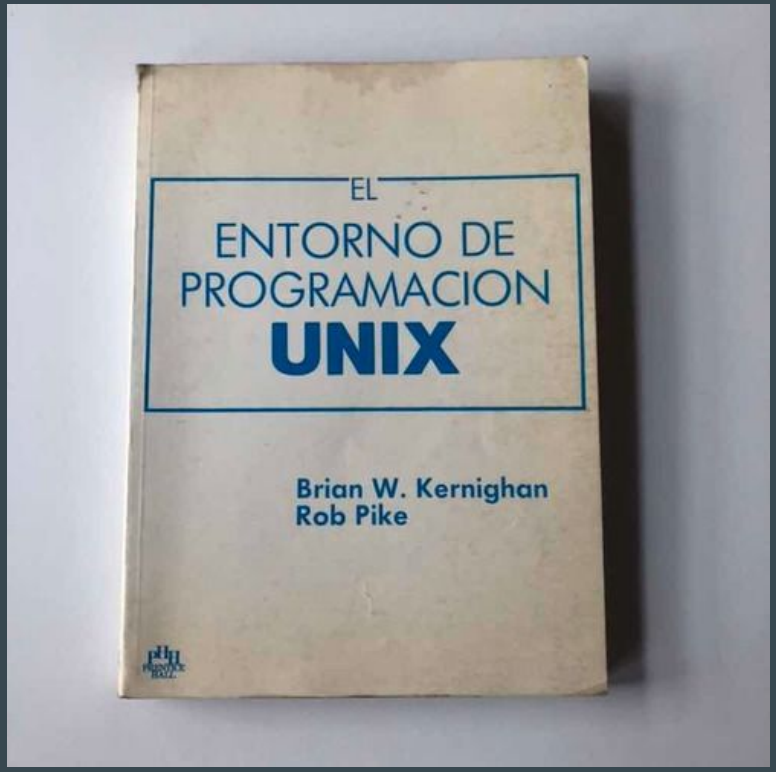
Manos a la obra.!!

Trabajar con copia_archivo.c



Esto tiene sus años..!!

En linux todo es un archivo.!!



El sistema de ficheros /proc

Contiene un sistema de archivos virtual. No existe físicamente en disco, sino que el kernel lo crea en memoria. Se utiliza para ofrecer información relacionada con el sistema (originalmente acerca de procesos, de aquí su nombre). Algunos de los archivos más importantes :

/proc/cpuinfo; Información acerca del procesador: tipo, marca, modelo, rendimiento, etc. (lscpu)

/proc/modules; Indica los módulos del núcleo que han sido cargados hasta el momento. (lsmod)

El sistema de ficheros /proc

Puede emplearse /proc como un mecanismo de interacción de propósito general entre el usuario y el kernel.

Los módulos pueden crear entradas /proc para interactuar con el usuario.!!

El sistema de ficheros /proc

Al leer/escribir de/en un “fichero” particular de este sistema (entrada /proc) se ejecuta una función del kernel que devuelve/recibe los datos. No ocupa espacio en disco.!!

Lectura: read callback (devolución de llamada al read del usuario)
Escritura: write callback (devolución de llamada al write del usuario)

En Linux, /proc muestra información de los procesos, uso de memoria, módulos, hardware, ...

También puede emplearse como mecanismo de interacción de propósito general entre el usuario y el kernel. Los módulos pueden crear entradas /proc para interactuar con el usuario.!!

Interfaz de operaciones de entrada /proc

file_operations Struct Reference

```
struct file_operations {
     struct module *owner;
     loff_t (*llseek) (struct file *, loff_t, int);
     ssize_t (*read) (struct file *, char *, size_t, loff_t *);
     ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
     int (*readdir) (struct file *, void *, filldir_t);
     unsigned int (*poll) (struct file *, struct poll_table_struct *);
     int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
     int (*mmap) (struct file *, struct vm_area_struct *);
     int (*open) (struct inode *, struct file *);
     int (*flush) (struct file *);
     int (*release) (struct inode *, struct file *);
     int (*fsync) (struct file *, struct dentry *, int datasync);
     int (*fasync) (int, struct file *, int);
     int (*lock) (struct file *, int, struct file_lock *);
     ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
     ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
};
```

Primera etapa cumplida...